字符串的模式匹配

学习目标

掌握模式匹配的含义

熟知BF算法

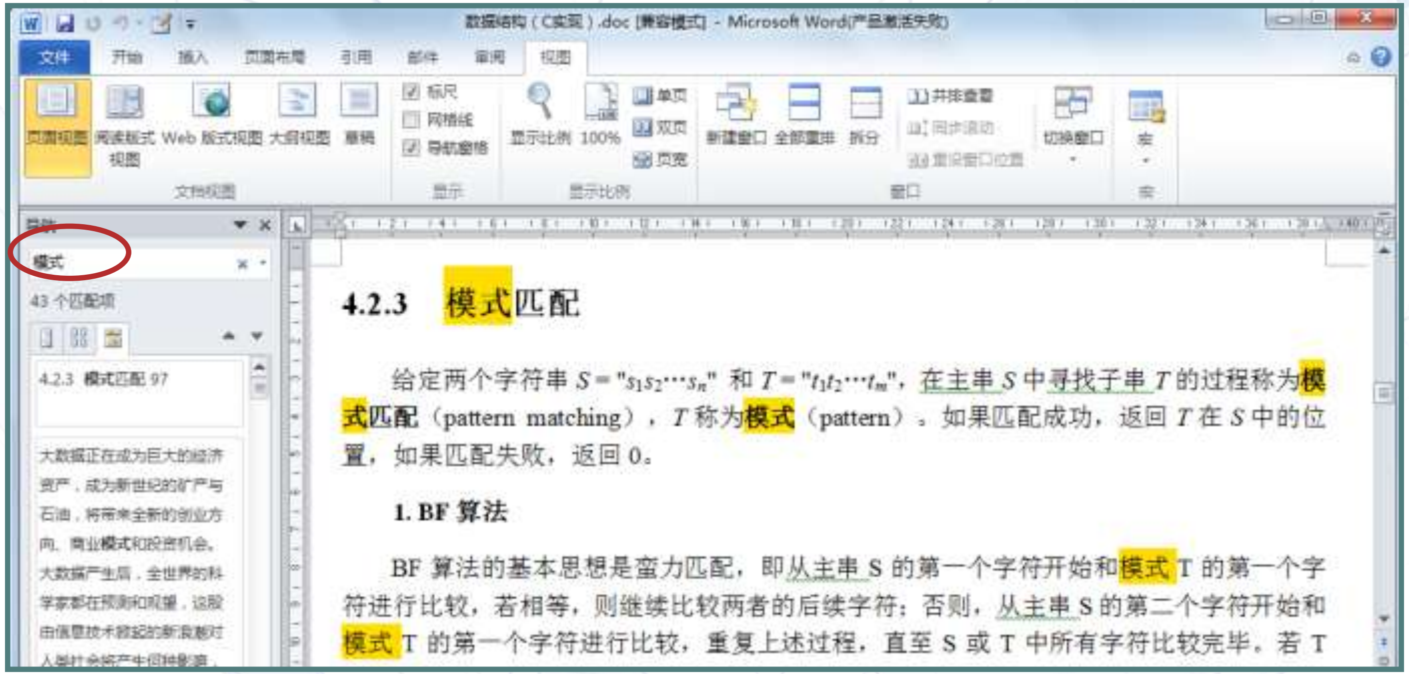
掌握KMP算法

什么是模式匹配

问题引入:模式匹配

- **查找功能**：文本编辑工具中，给定一段文本，用户提供特定的关键词，找出这个关键词在文本中出现的位置，就是字符串匹配问题。

输入关键词 → 字符串匹配算法 → 定位关键词位置



字符串的模式匹配

概念：在字符串 *s* 中找出与字符串 *t* 相等的子串的操作称为字符串的模式匹配，又称为子串的定位操作。

- 如果匹配成功，返回 *t* 在 *s* 中的位置；否则返回 0/-1/NIL
- 其中字符串 *s* 称为主串或目标串，字符串 *t* 称为模式串。

字符串的模式匹配

形式化描述：

假设目标串***s***使用一个长度为n的字符数组s[0,1,...,n-1]表示，模式串 ***t*** 使用一个长度为m (m≤ n)的数组t[0,1,...,m-1]表示，如果存在***p*** (0 ≤ p ≤ n - m)，使得s[p+0, p+1, ... , p+m-1] = t[0, 1, ..., m-1]，则***p***被称为一个有效位移。字符串匹配就是从字符串***s***中找出存在的有效位移***p***。

模式匹配问题有什么特点?

（1） 算法的一次执行时间：问题规模通常很大，常常在大量信息中进行匹配

（2） 算法改进所取得的积累效益：模式匹配操作经常被调用，执行频率高

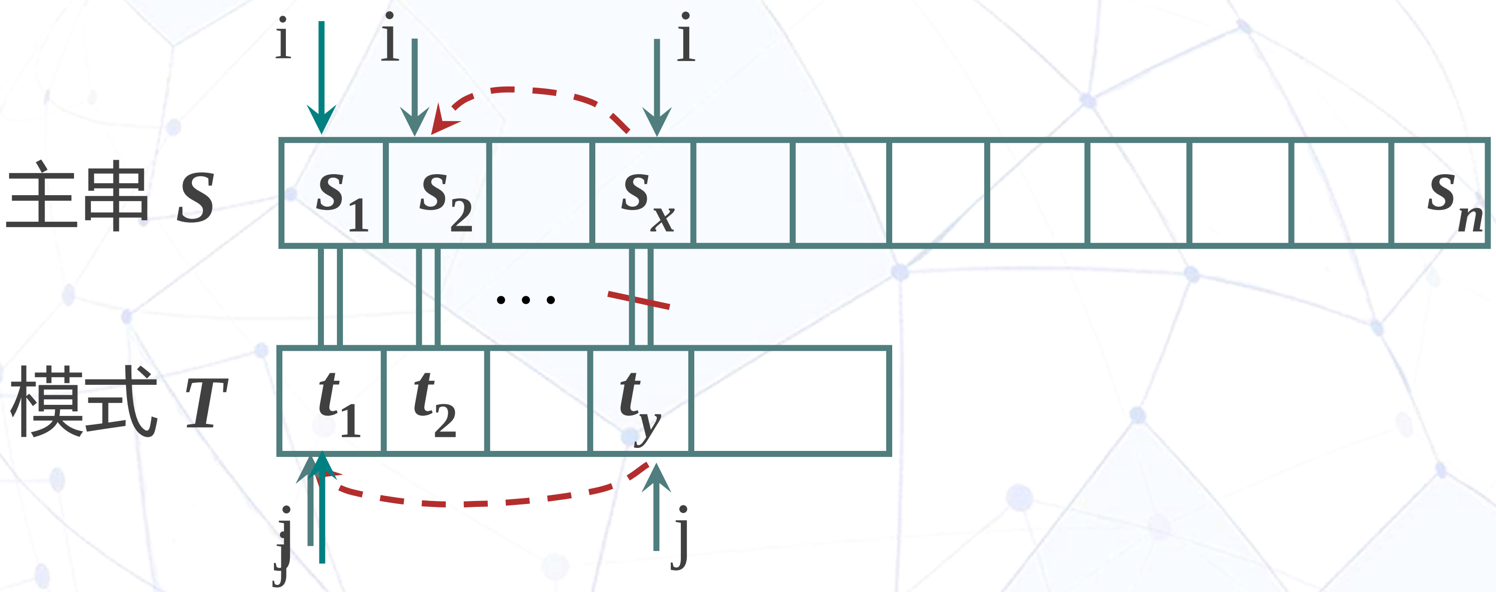
模式匹配算法：朴素字符串匹配算法（BF算法）、KMP算法、BM算法、KR算法、Sunday算法等。

BF算法

朴素模式匹配算法是字符串模式匹配算法中最简单的暴力解法，又称为BF（Brute Force）算法。

实现：枚举每个目标串 ***s*** 与模式串 ***t*** 等长的子串，判断是否匹配

BF算法——基本思想

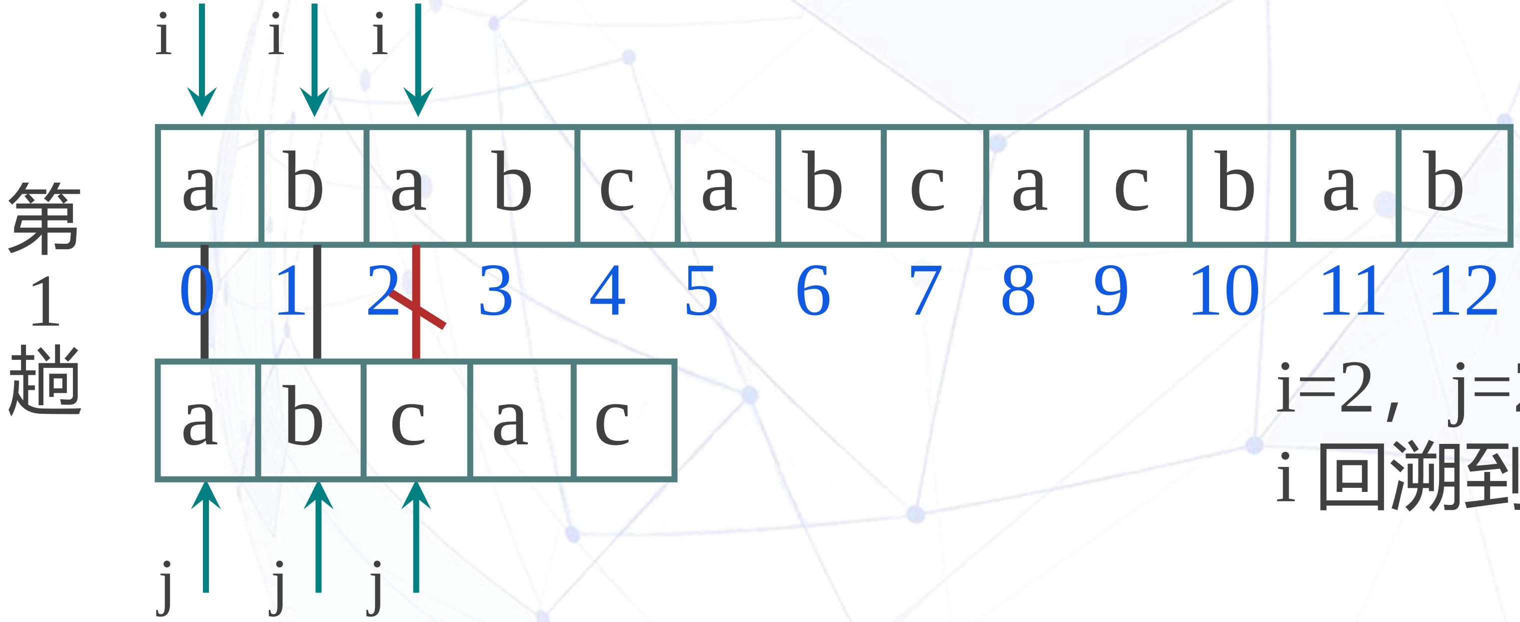


1. 在串 S 和串 T 中设比较的起始下标 i 和 j；
2. 循环直到 S 或 T 的所有字符均比较完
   2.1 如果S[i] 等于T[j]，继续比较 S 和 T 的下一个字符；
   2.2 否则，将 i 和 j **回溯**，准备下一趟比较；
3. 如果T中所有字符均比较完，则返回匹配的起始比较下标；否则返回 0；

如何回溯?
i? j?

BF算法——运行实例

例：主串 S = "ababcabcacbab"， 模式 T ="abcac"



i=2，j=2失败；
i 回溯到 1，j 回溯到 0

BF算法——运行实例

例：主串 S = "ababcabcacbab"，模式 T ="abcac"

第2趟

	i											
a	b	a	b	c	a	b	c	a	c	b	a	b
0	1	2	3	4	5	6	7	8	9	10	11	12

a	b	c	a	c
j				

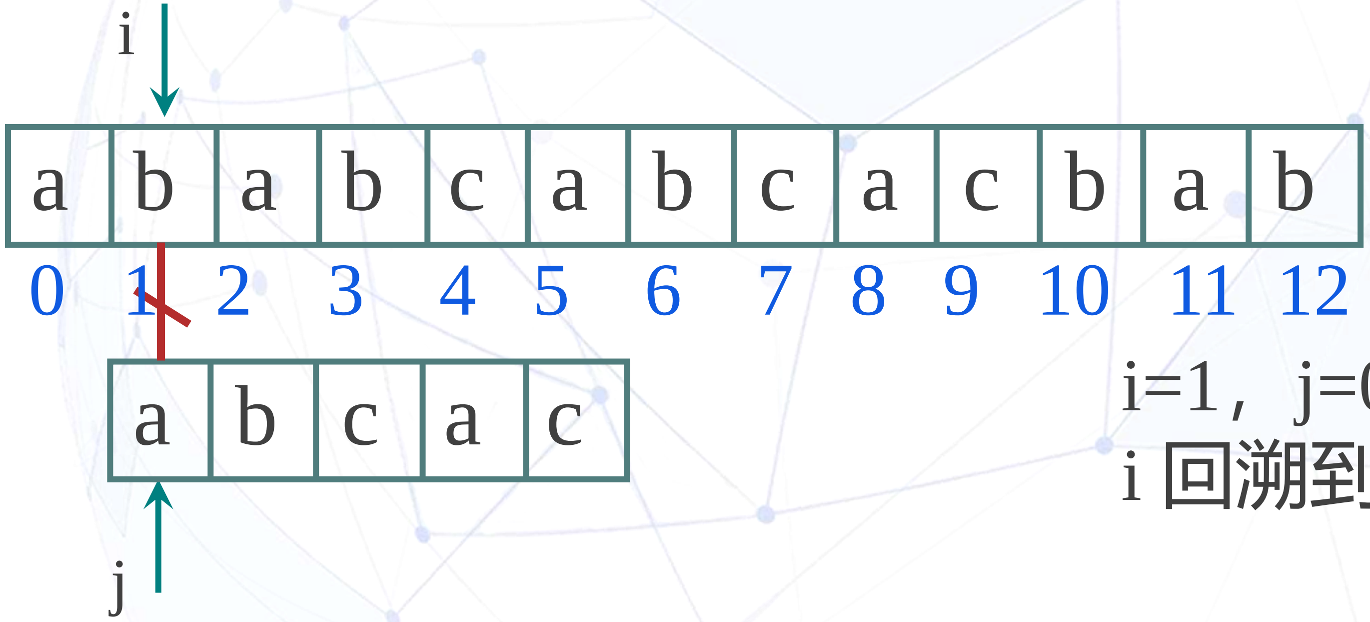
i=2，j=2失败；

i 回溯到 1，j 回溯到 0

BF算法——运行实例

例：主串 S = "ababcabcacbab"，模式 T ="abcac"

第2趟



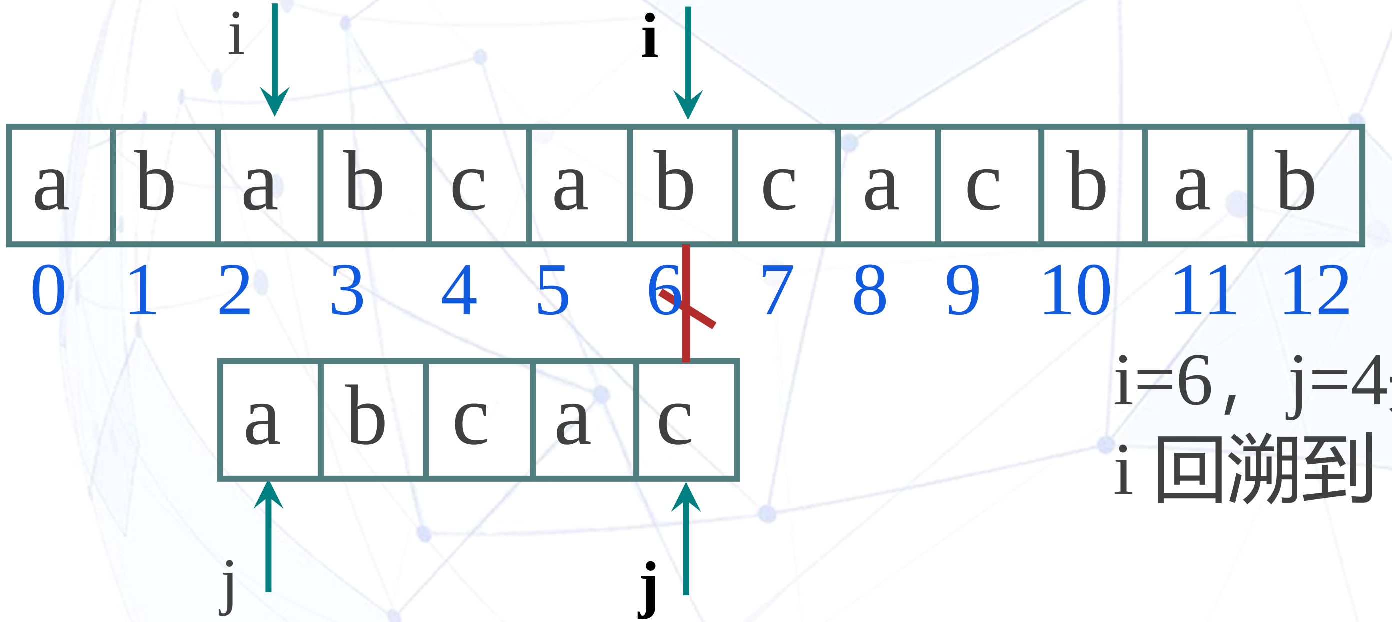
i=1，j=0失败

i 回溯到 2，j 回溯到 0

BF算法——运行实例

例：主串 S = "ababcabcacbab"， 模式 T ="abcac"

第
3
趟

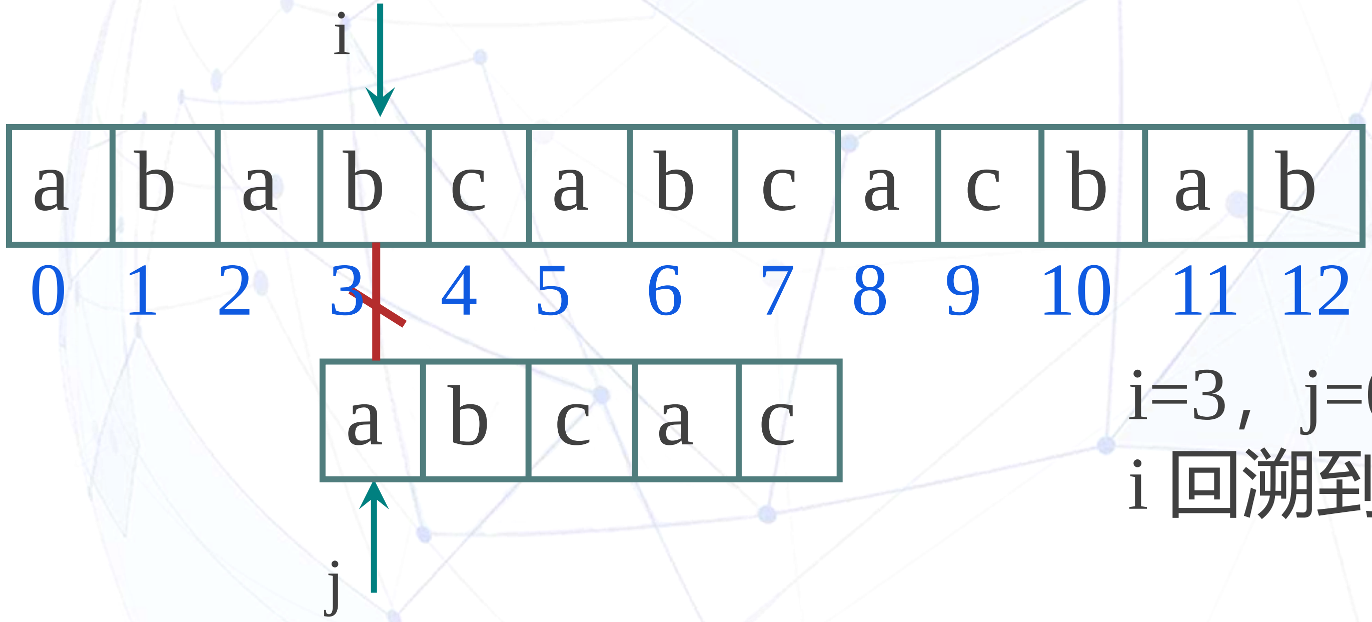


i=6，j=4失败
i 回溯到 3，j 回溯到 0

BF算法——运行实例

例：主串 S = "ababcabcacbab"，模式 T ="abcac"

第4趟

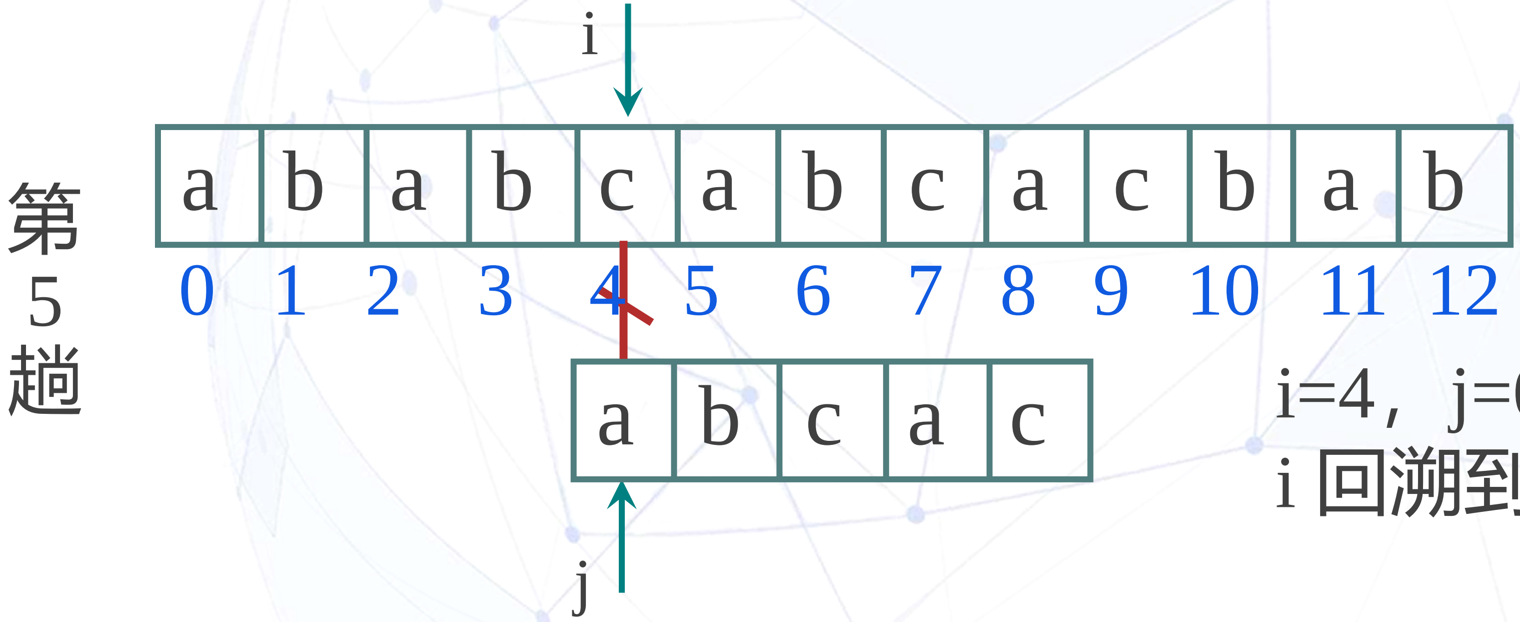


i=3，j=0失败

i 回溯到 4，j 回溯到 0

BF算法——运行实例

例：主串 S = "ababcabcacbab"， 模式 T ="abcac"

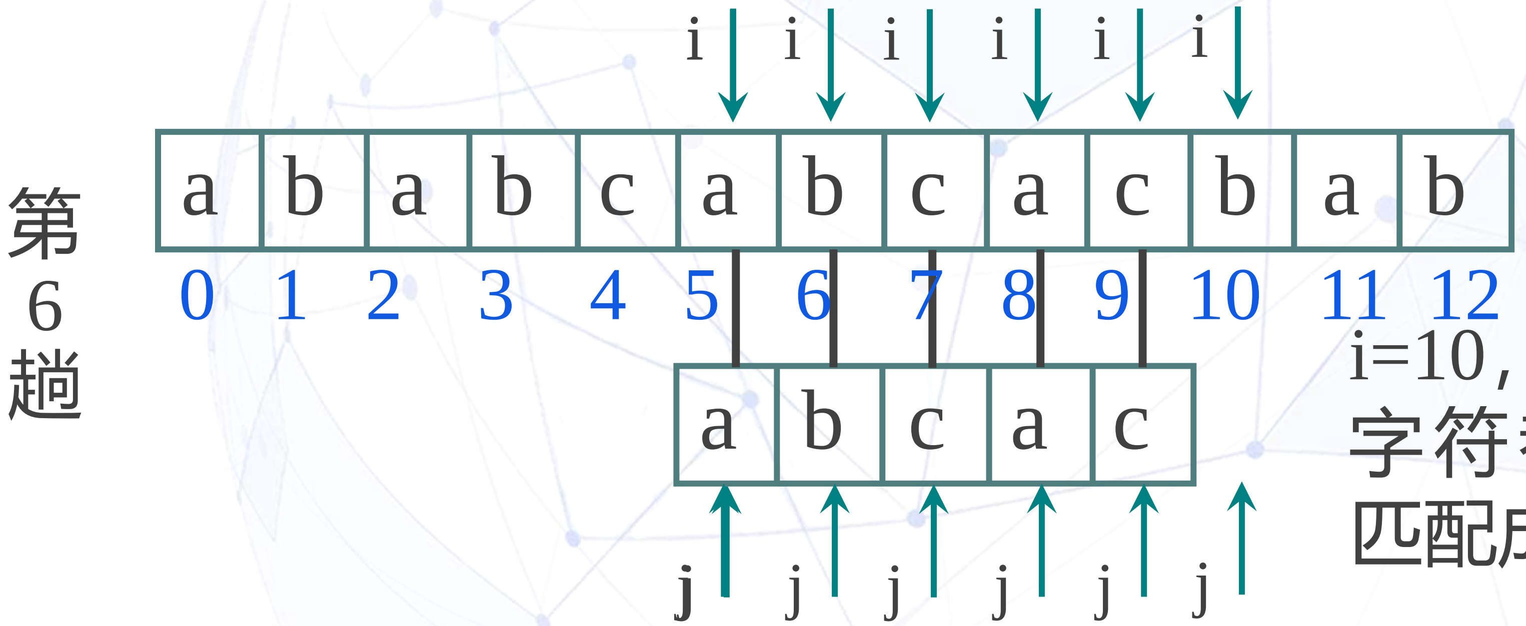


i=4，j=0失败

i 回溯到 5，j 回溯到 0

BF算法——运行实例

例：主串 S = "ababcabcacbab"，模式 T ="abcac"



i=10，j=5，T中全部字符都比较完毕，匹配成功

BF算法——算法描述

```
int BF(char S[ ], char T[ ])
{
    int i = 0, j = 0;
    while (S[i] != '\0'&&T[j] != '\0')
    {
        if (S[i] == T[j]) {
            i++;   j++;
        }
        else {
            i = i – j + 1;   j = 0;
        }
    }
    if (T[j] == '\0') return (i – j + 1);
    else return 0;
}
```

```
int BF(char S[ ], char T[ ])
{
    int i = 0, j = 0, start = 0;
    while (S[i] != '\0'&&T[j] != '\0')
    {
        if (S[i] == T[j]) {
            i++;   j++;
        }
        else {
            start++;  i = start;  j = 0;
        }
    }
    if (T[j] == '\0') return start + 1;
    else return 0;
}
```

BF算法——性能分析

$$S = "s_1 s_2 \dots s_n" \qquad T = "t_1 t_2 \dots t_m"$$

在**匹配不成功**（即S不包含T）的情况下：

S_1到S_{n-m+1}，都会与T进行m次比较，所以共比较(n-m+1)*m，时间复杂度达到O(n*m)

BF算法——性能分析

$$S = "s_1 s_2 \ldots s_n" \qquad T = "t_1 t_2 \ldots t_m"$$

在**匹配成功**（即S包含T）的情况下，考虑两种极端情况：

最好情况：不成功的匹配都发生在串 T 的第 1 个字符

例如：$S = "aaaaaaaaaabcd"$

$T = "bcd"$

设匹配成功发生在 s_i 处，则前面i-1次首字母比较i-1次，第i次匹配成功需要比较m次，所以需要比较i-1+m次，因为i可能发生在n-m+1个位置，假设等概率，平均比较次数为：

$$\sum_{i=1}^{n-m+1} p_i \times (i-1+m) = \sum_{i=1}^{n-m+1} \frac{1}{n-m+1} \times (i-1+m) = \frac{(n+m)}{2} = O(n+m)$$

BF算法——性能分析

$$S = "s_1 s_2 \ldots s_n" \qquad T = "t_1 t_2 \ldots t_m"$$

在**匹配成功**（即S包含T）的情况下，考虑两种极端情况：

最坏情况：不成功的匹配都发生在串 T 的最后一个字符

例如：$S = "aaaaaaaaaaaabccccc"$

$T = "aaab"$

设匹配成功发生在 s_i 处，则：

$$\sum_{i=1}^{n-m+1} p_i \times (i \times m) = \sum_{i=1}^{n-m+1} \frac{1}{n-m+1} \times (i \times m) = \frac{m(n-m+2)}{2} = O(m \times n)$$

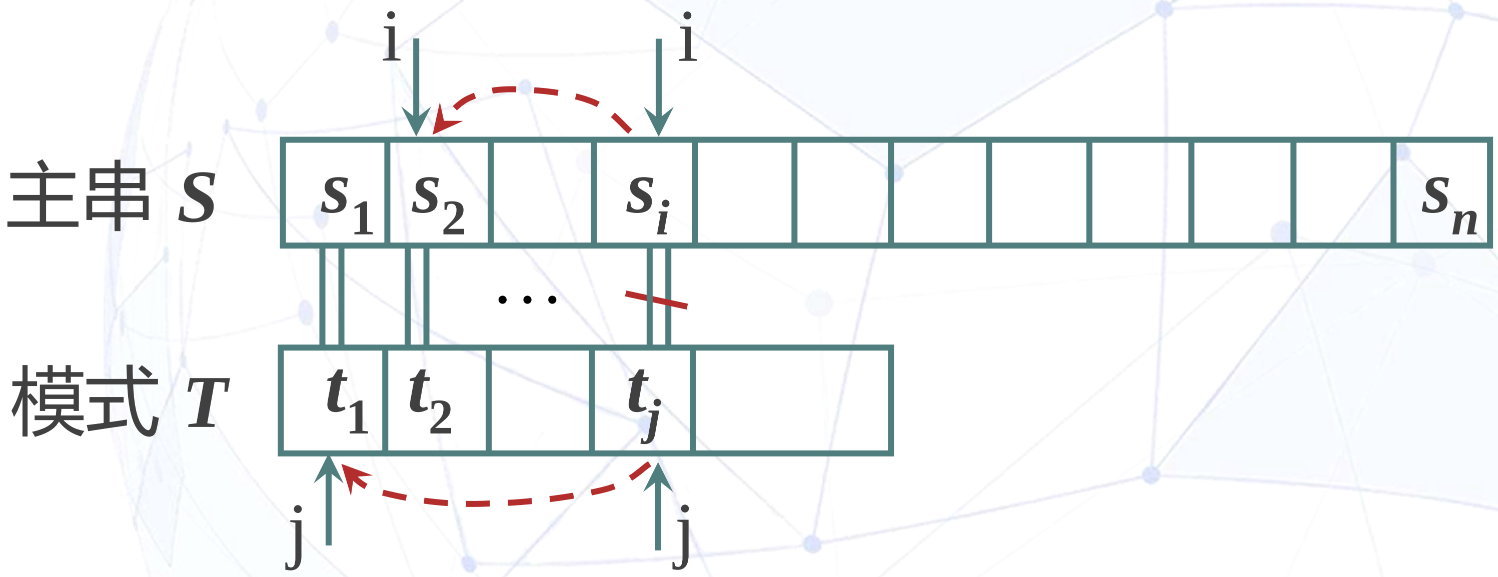
模式匹配KMP算法

KMP算法

- **BF算法缺点**：当模式串失配时将其整体向后移动一位，然后从头开始匹配，将目标串中每个位置作为起点与模式串中的字符进行逐个比较会耗费较长时间。
- **提出**：KMP算法由D.E.Knuth，J.H.Morris和V.R.Pratt提出的，因此也称为克努特—莫里斯—普拉特算法。
- **主要思想**：假设BF算法在失配时已经匹配到了模式串的第 j 位，则说明目标串与模式串的前 j-1 位是匹配成功的，利用已匹配的信息可对BF算法进行优化。

分析BF算法

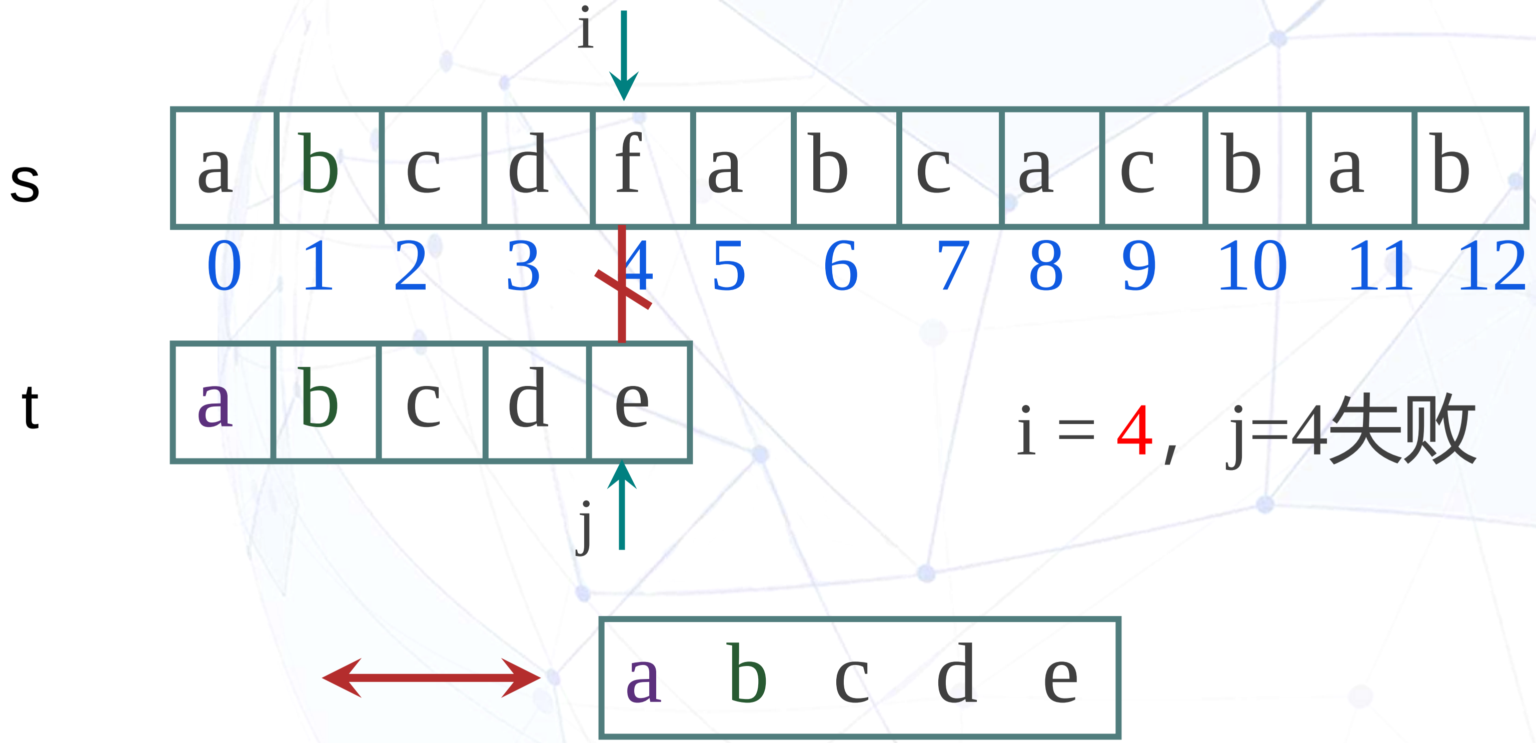
为什么BF算法的性能较低?



在每趟匹配不成功时存在大量回溯，没有利用已经部分匹配的结果

比如这个例子：

t中每个字符都不相同，如果在i位置发生不匹配，所以$t_0 \sim t_{i-1} = s_0 \sim s_{i-1}$，
因每个字符都不相同，t_0不会等于$s_1 \sim s_{i-1}$中的任一个



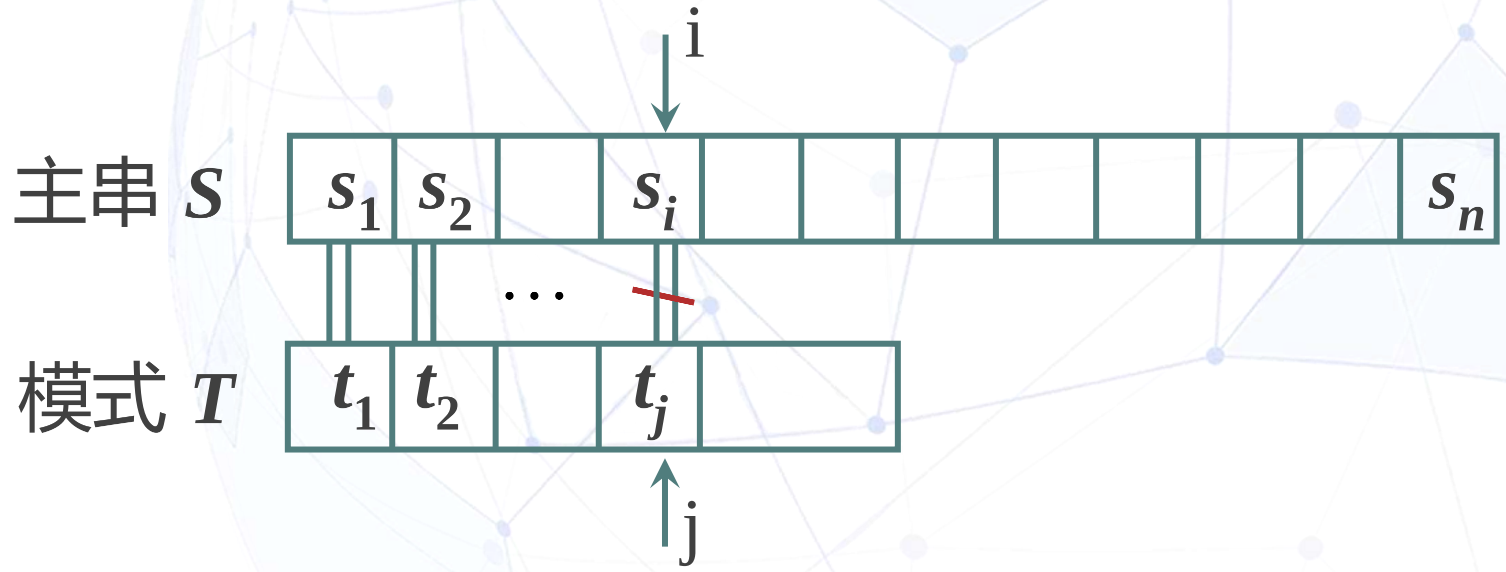
i = 4， j=4失败

i可以不回溯，j可以从0开始，即可以利用前面的匹配结果可以帮助下次匹配

分析BF算法

如何在匹配不成功时主串不回溯?

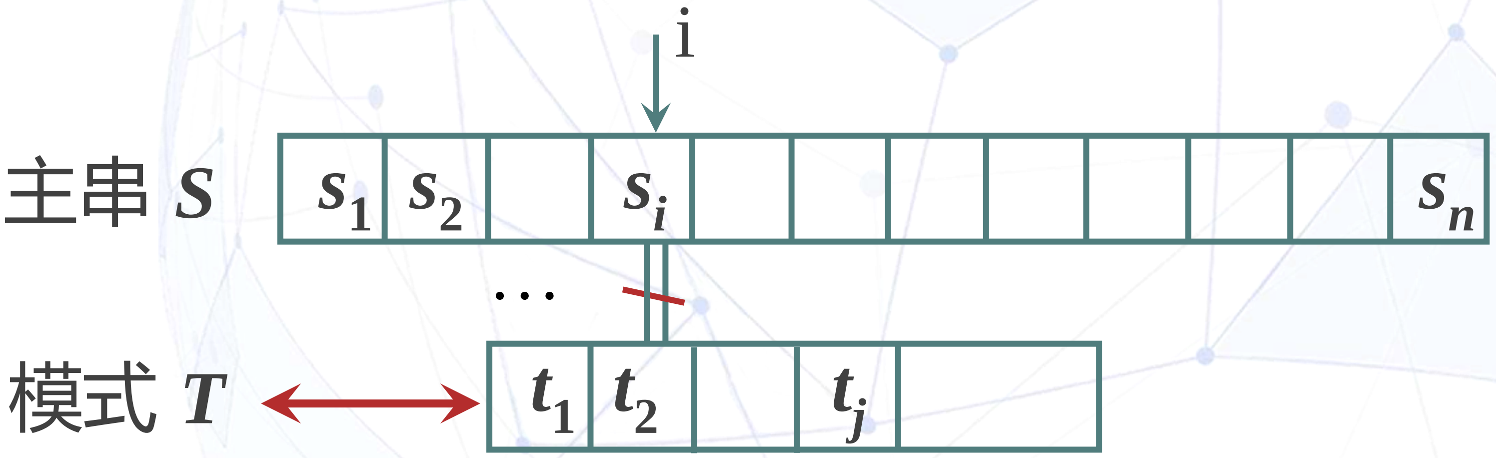
主串不回溯，模式就需要向右滑动一段距离



分析BF算法

如何在匹配不成功时主串不回溯?

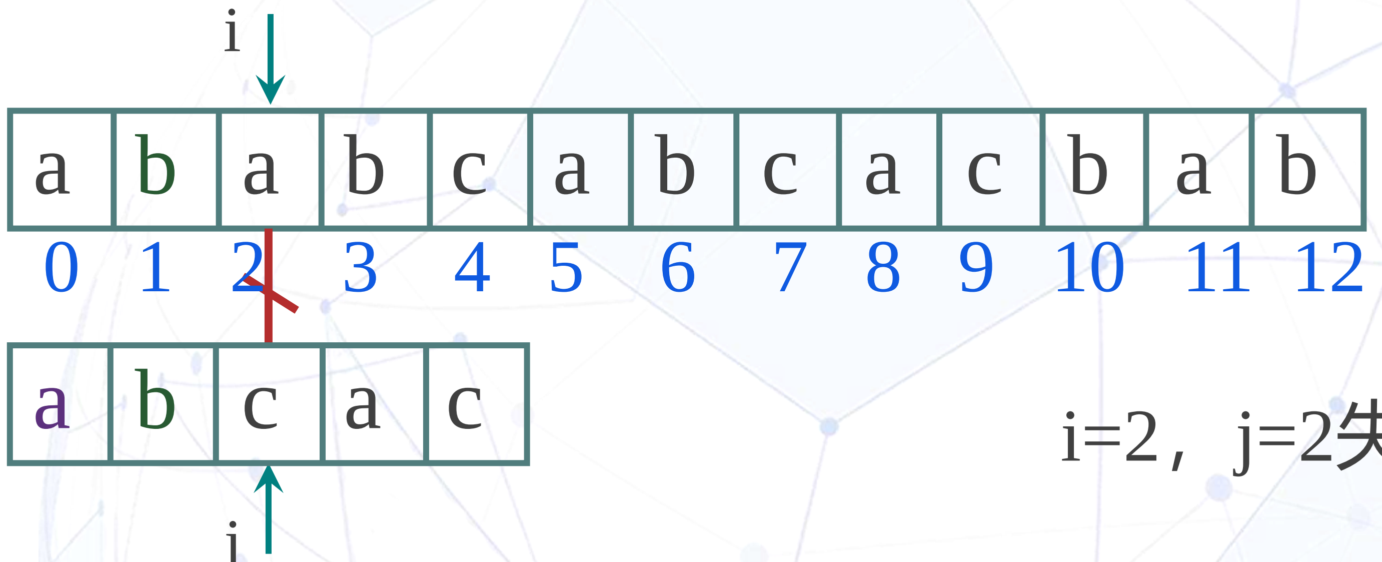
主串不回溯，模式就需要向右滑动一段距离



如何确定模式的滑动距离?

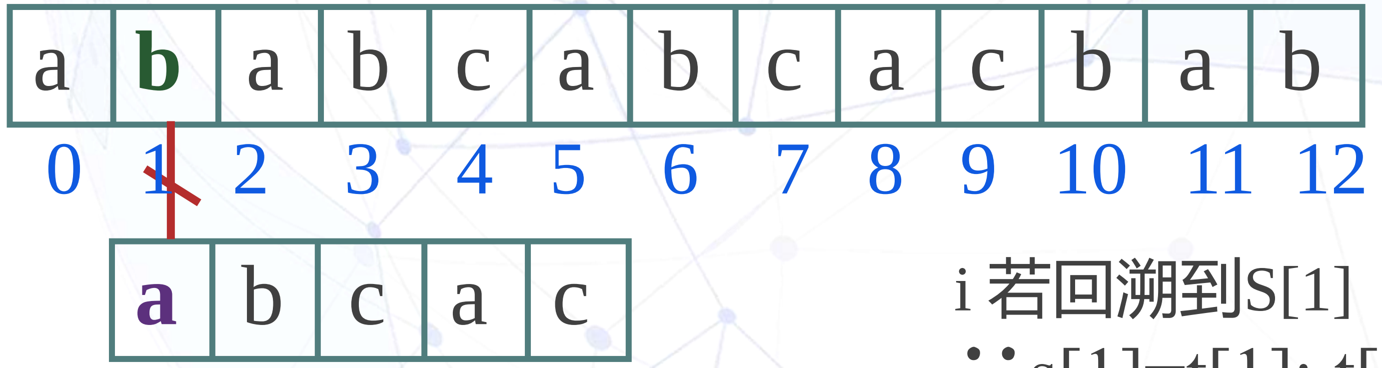
分析BF算法

第1趟



i=2，j=2失败；

第2趟

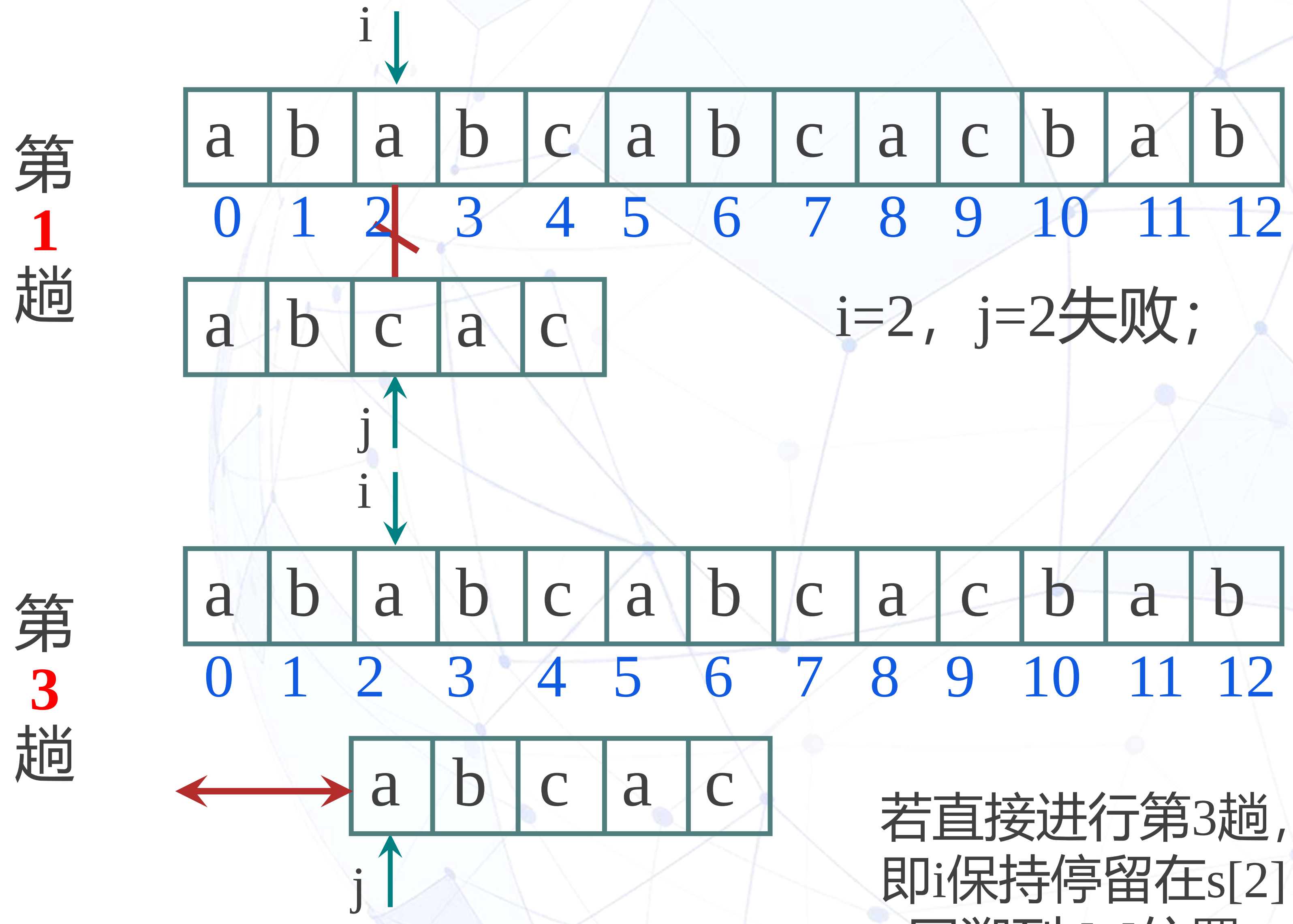


i 若回溯到S[1]

∵ s[1]=t[1]; t[0]≠t[1]

∴ t[0]≠s[1]，这步实际无需进行

分析BF算法



i=2，j=2失败；

若直接进行第3趟，
即i保持停留在s[2]，不回溯
j 回溯到t[0]位置

分析BF算法

第3趟

i

a	b	a	b	c	a	b	c	a	c	b	a	b
0	1	2	3	4	5	6	7	8	9	10	11	12

a	b	c	a	c

j

i = 6，j=4失败

第4趟

a	b	a	b	c	a	b	c	a	c	b	a	b
0	1	2	3	4	5	6	7	8	9	10	11	12

a	b	c	a	c

i若回溯到s[3]，j到0

t[0]≠s[3] , 这一趟也可以跳过

分析BF算法

第3趟

i

a	b	a	b	c	a	b	c	a	c	b	a	b
0	1	2	3	4	5	6	7	8	9	10	11	12

a	b	c	a	c

j

i=6，j=4失败

第5趟

a	b	a	b	c	a	b	c	a	c	b	a	b
0	1	2	3	4	5	6	7	8	9	10	11	12

a	b	c	a	c

若i回溯到s[4]

t[0]≠s[4], 这一趟也可以跳过

分析BF算法

第3趟

i

a	b	a	b	c	a	b	c	a	c	b	a	b
0	1	2	3	4	5	6	7	8	9	10	11	12

a	b	c	a	c

j

i=6，j=4失败

第6趟

i

a	b	a	b	c	a	b	c	a	c	b	a	b
0	1	2	3	4	5	6	7	8	9	10	11	12

a	b	c	a	c

j

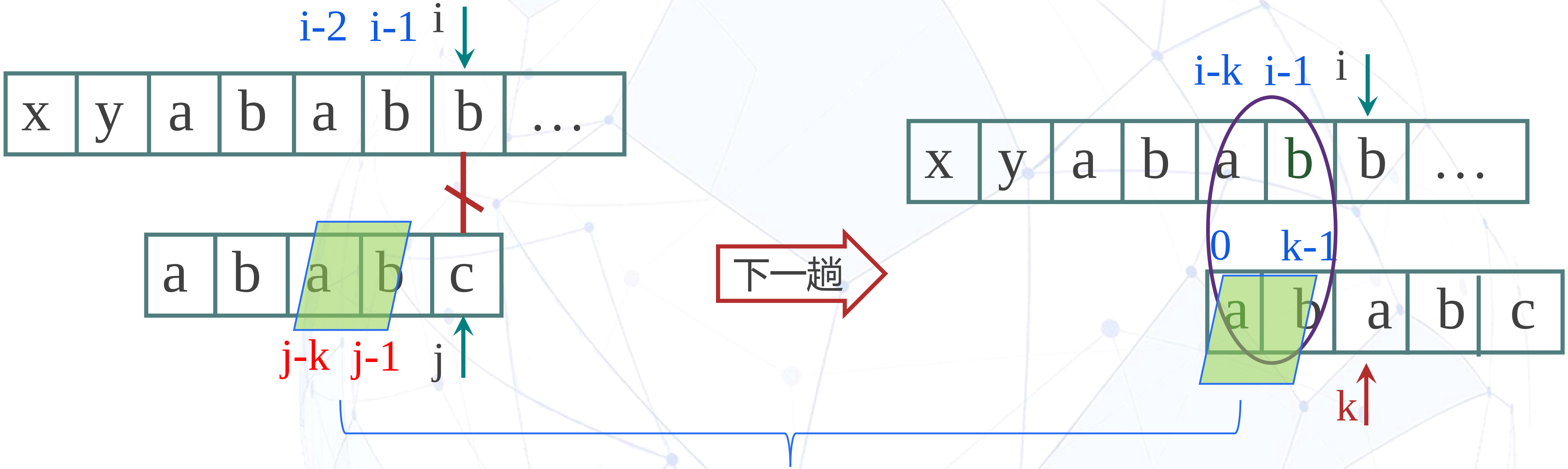
若i回溯到S[5], 匹配成功。
此时t从s[2]移动到s[5],
此时s[5]=t[0]
若 j 从t[1]开始比较，i 似乎可不回溯

KMP算法

结论： 主串的i 可以不回溯，模式串向右滑动到新的比较起点 k

如何由当前部分匹配结果确定模式向右滑动的新比较起点 k?

假设这次比较 j 可以从 k=2 开始



即此时满足（1）**T[0]~T[k-1] = S[i-k]~S[i-1]**

那么什么时候 j 可以从 k=2 开始呢?

在i和j不匹配时，前面的T[j-k]…T[j-1]必然是与S[i-k]~S[i-1]匹配的，如果T[0]-T[k-1]与T[j-k]…T[j-1]是相同的，就不需要再进行这块的比较，所以还需要满足下面一个条件：

（2）**T[j-k]~T[j-1] = S[i-k]~S[i-1]**

KMP算法

结论： i 可以不回溯，模式向右滑动到新的比较起点 k

如何由当前部分匹配结果确定模式向右滑动的新比较起点 k?

（1）T[0]~T[k-1] = S[i-k]~S[i-1]

（2）T[j-k]~T[j-1] = S[i-k]~S[i-1]

$\boxed{\text{T[0]\textasciitilde T[k-1] = T[j-k]\textasciitilde T[j-1]}}$

KMP算法

- T[0] ~ T[k-1] = T[j-k] ~ T[j-1] 说明了什么?

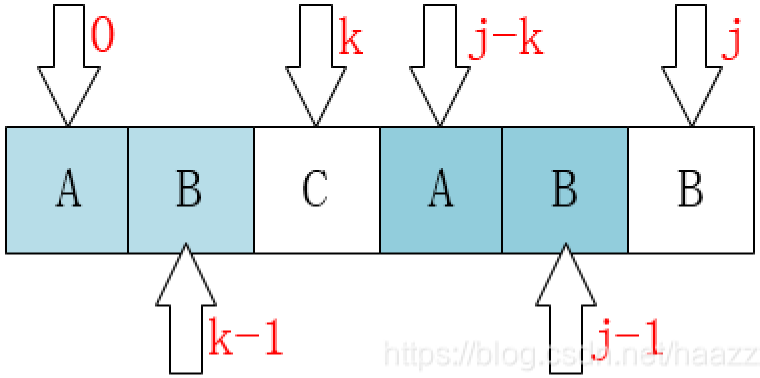
（1） k 与 j 具有函数关系，由当前失配位置 j ，可以计算出滑动位置 k

（2）滑动位置 k 仅与模式串 T 有关

- T[0] ~ T[k-1] = T[j-k] ~ T[j-1] 的物理意义是什么?

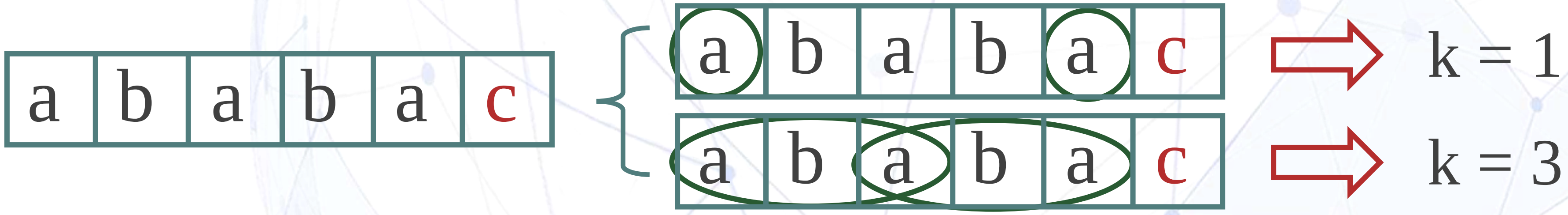
长度为 k 的前缀

长度为 k 的后缀

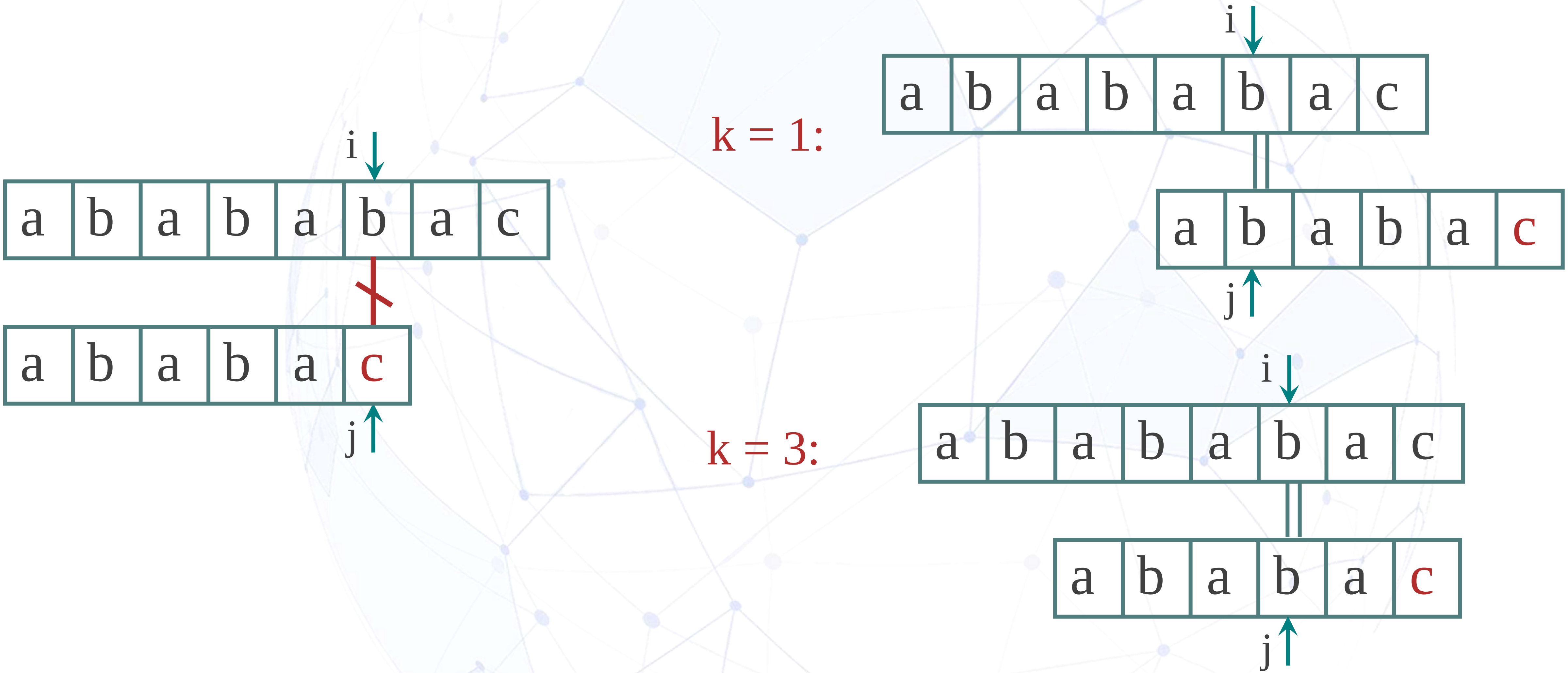


KMP算法

T[0] ~ T[j]中前缀和后缀相等的真子串唯一吗?



KMP算法 模式应该向右滑多远才能保证算法的正确性?



max {*k* | 1≤*k* < *j* 且T[0] … T[k-1] = T[j-k] … T[j-1]}

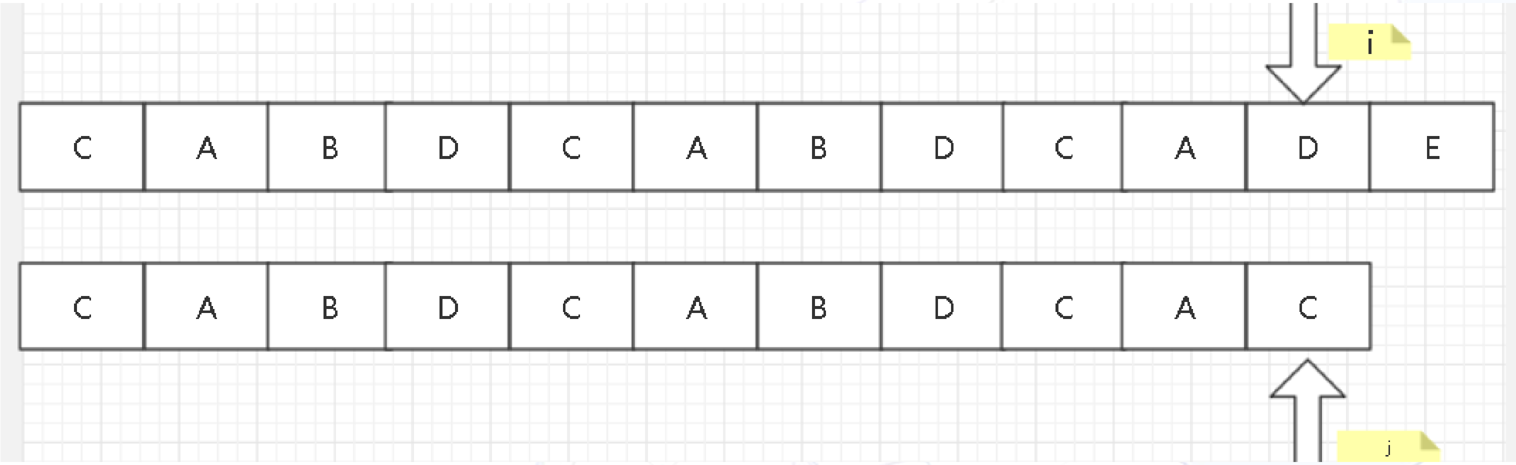
最长公共前后缀

公共前后缀：字符串的前缀集合与后缀集合中相同的子串。

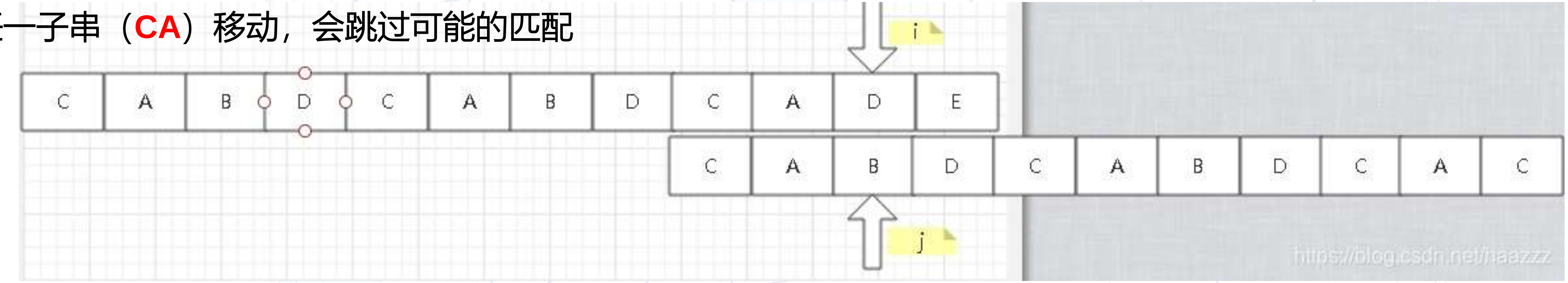
最长公共前后缀：字符串的前缀集合与后缀集合中相同的长度最长的子串。

例：字符串" aabaa "

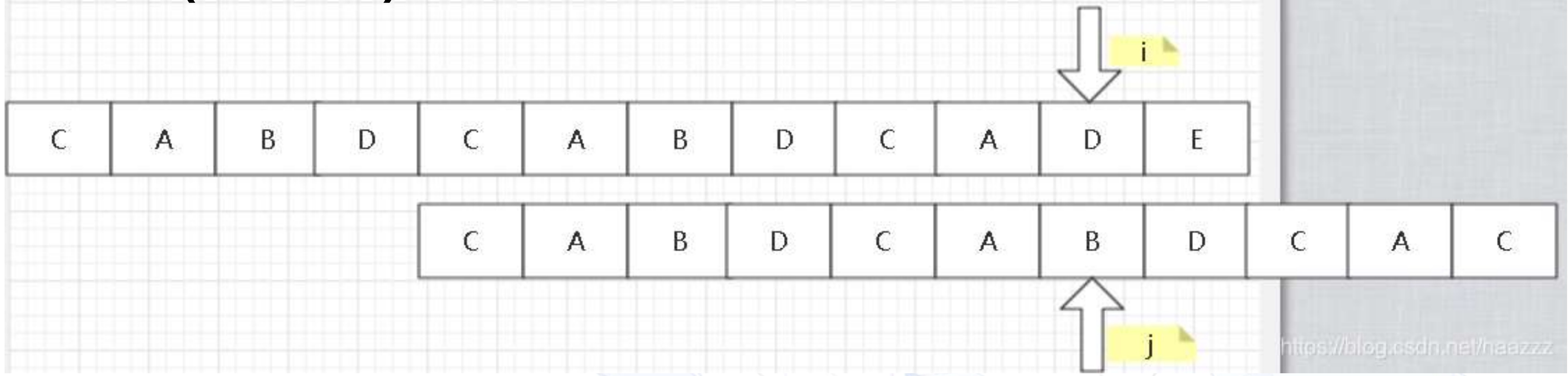
前缀集合为{"a","aa","aab","aaba"}，后缀集合为{"a","aa","baa","abaa"}，其公共前后缀包括"a"和"aa"两个子串，最长公共前缀为"aa"这个子串。



按任一子串（CA）移动，会跳过可能的匹配



按最长公共子串(CABDCA)移动，则不会跳过可能的匹配



运行实例

设next[j]表示在匹配过程中与T[j]比较不相等时，下标 j 的回溯位置

$$
next[j]=\begin{cases} -1 & j=0 \\ \max\{k \mid 1\le k< j \text{ 且} T[0]\dots T[k-1]=T[j-k]\dots T[j-1]\} & \text{集合非空} \\ 0 & \text{其它情况} \end{cases}
$$

下标：	0	1	2	3	4
模 式 串 T：	a	b	a	b	c
k = next[j]：	-1	0	0	1	2

j=0时, k = -1

j=1时, k = 0

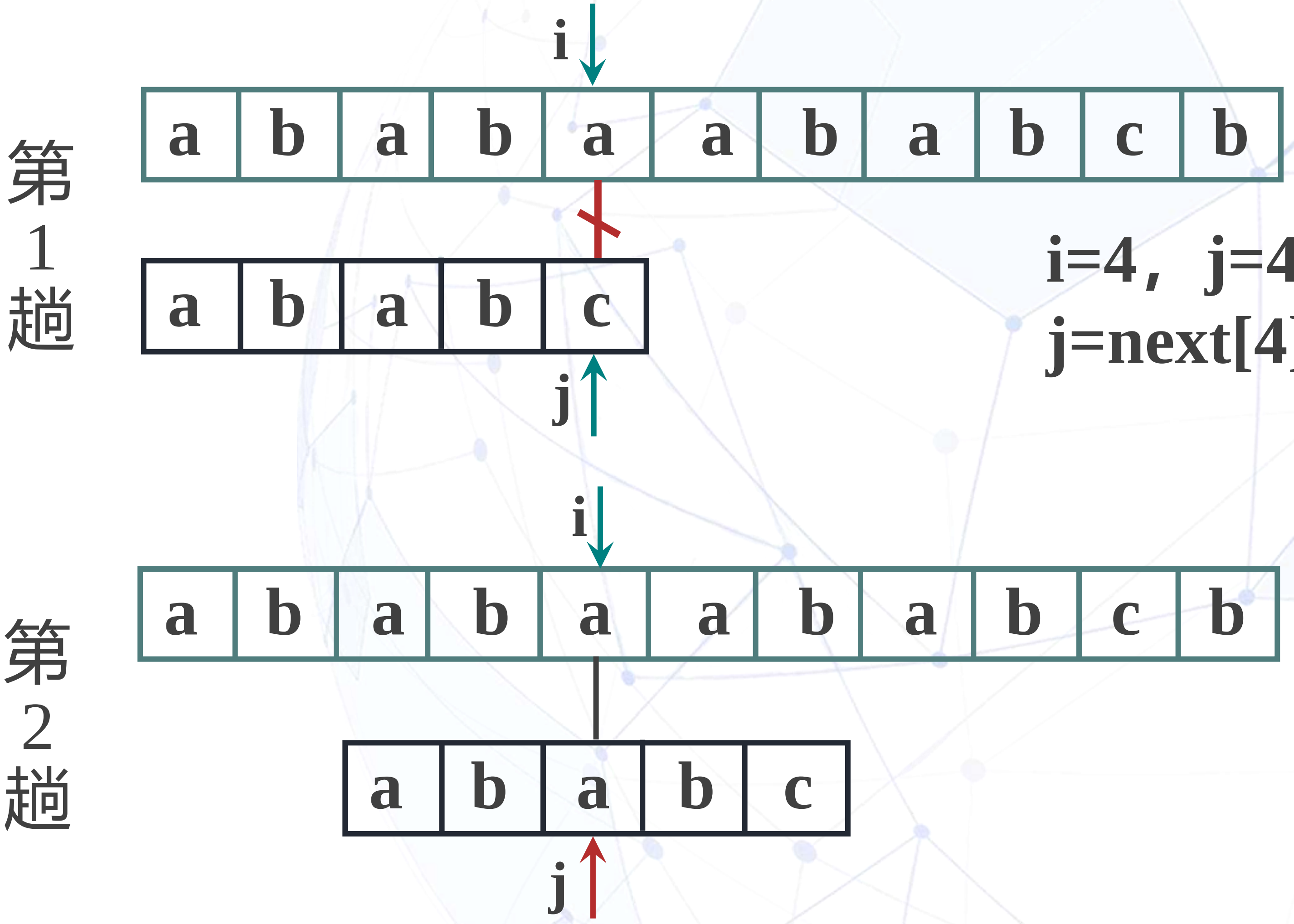
j=2时, T[0]≠T[1]，因此，k = 0

j=3时, T[0] = T[2]，T[0]T[1] ≠T[1]T[2]，因此，k = 1

j=4时, T[0] ≠ T[3]，T[0]T[1] = T[2]T[3]，T[0]T[1]T[2]≠T[1]T[2]T[3]，因此，k = 2

运行实例

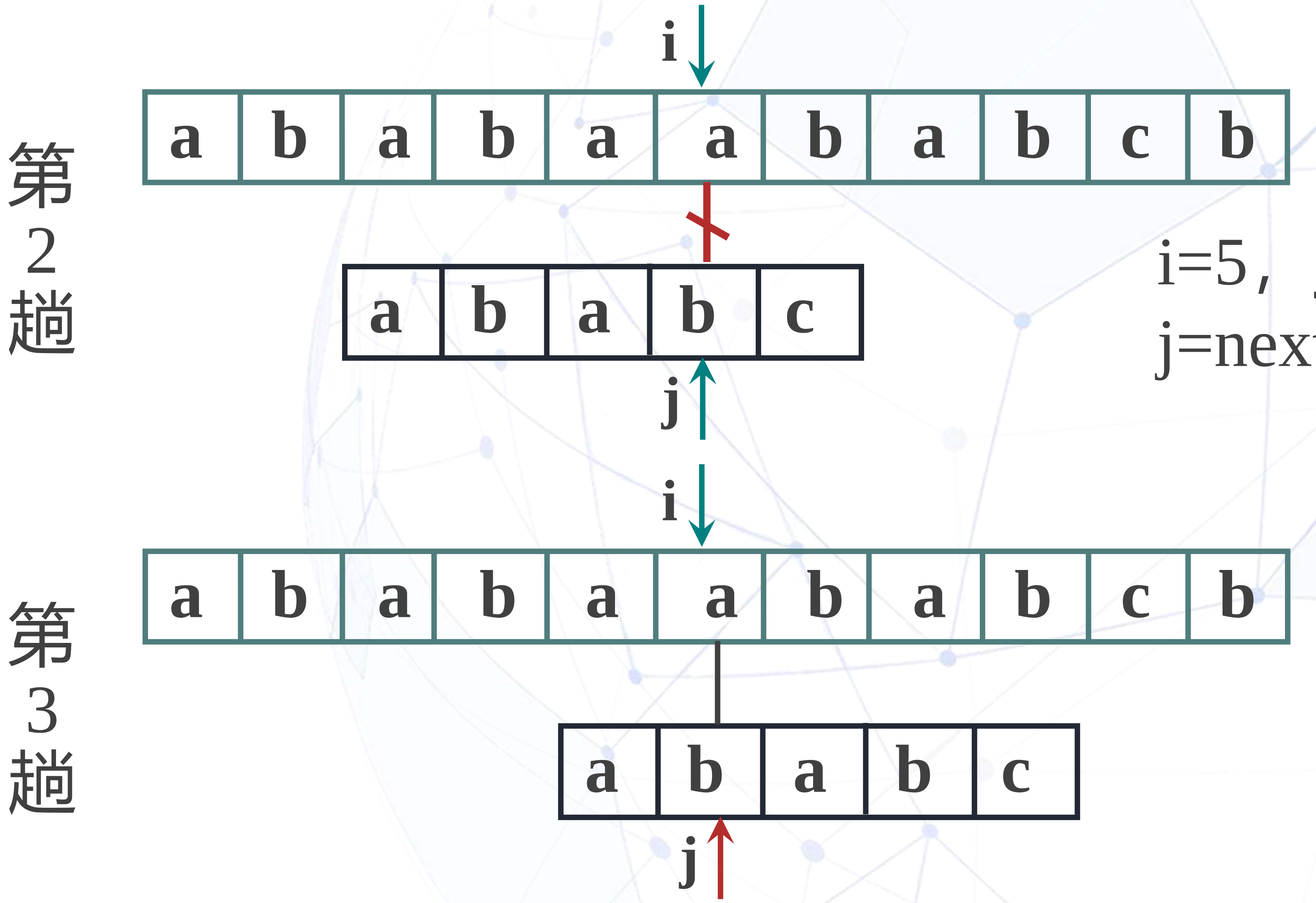
next[j]={-1, 0, 0, 1, 2}



i=4，j=4失败;
j=next[4]=2

运行实例

next[j]={-1, 0, 0, 1, 2}



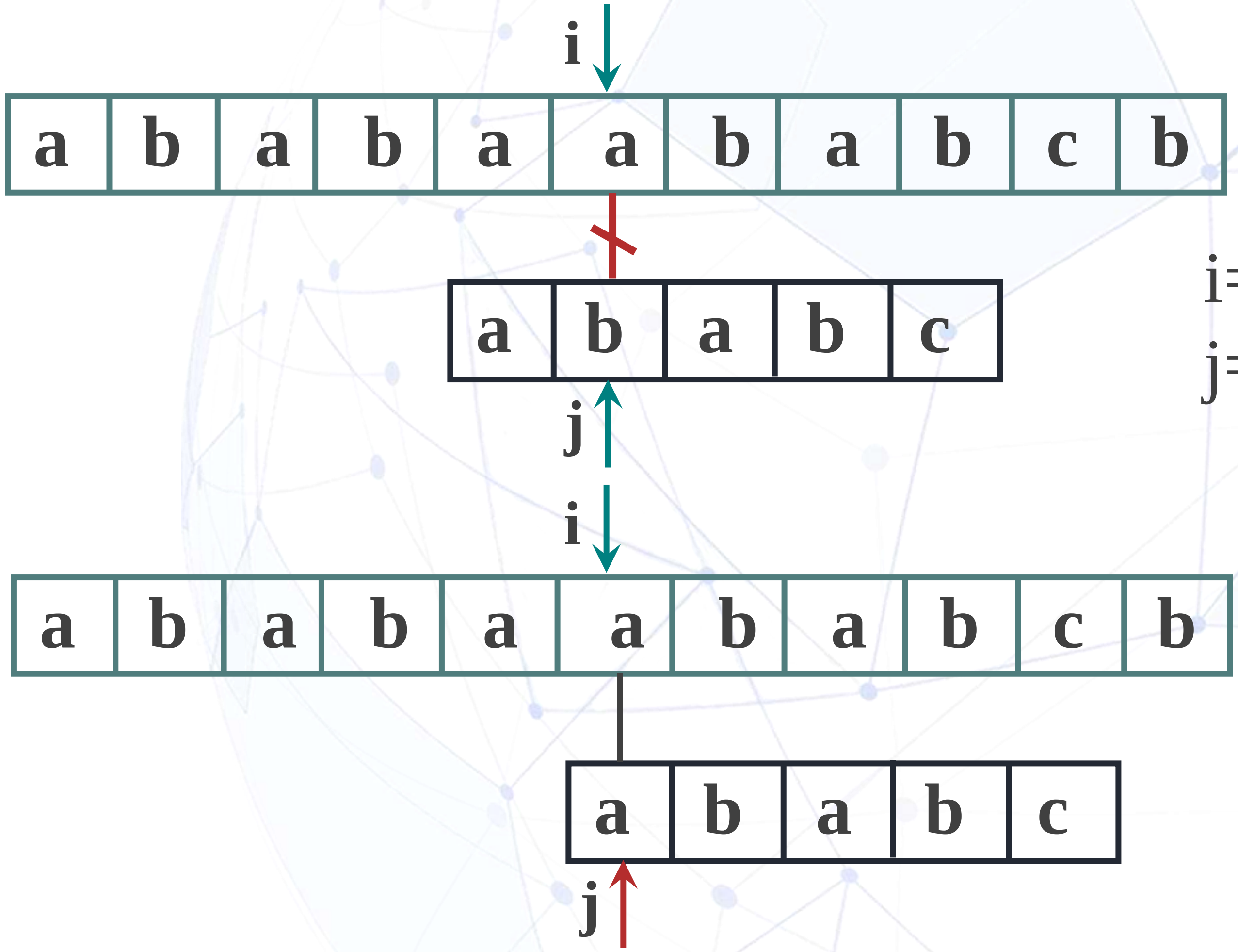
i=5，j=3失败;
j=next[3]=1

运行实例

next[j]={-1, 0, 0, 1, 2}

第3趟

第4趟



i=5，j=1失败;
j=next[1]=0

next函数手算

下标:	0	1	2	3	4	5
模式串 T:	a	b	c	a	b	d
k = next[j]:	-1	0	0	0	1	2

next函数手算

下标:	**0**	1	2	3	4	5
模 式 串 T:	a	b	a	b	a	a
k = next[j]:	-1	0	0	1	2	3

next函数手算

下标:	0	1	2	3	4
模 式 串 T:	a	a	a	a	b
k = next[j]:	-1	0	1	2	3

KMP算法

1. 在串 S 和串 T 中分别设比较的起始下标 i 和 j；

2. 循环直到 S 或 T 的所有字符均比较完

   2.1 如果S[i]等于T[j]，继续比较 S 和 T 的下一个字符；

   否则，将 j 向右滑动到next[j]位置，即 j=next[j]；

   2.2 如果 j=-1，则将 i 和 j 分别加 1，准备下一趟比较；

3. 如果 T 中所有字符均比较完毕，则返回匹配的起始下标；否则返回 0；

KMP算法

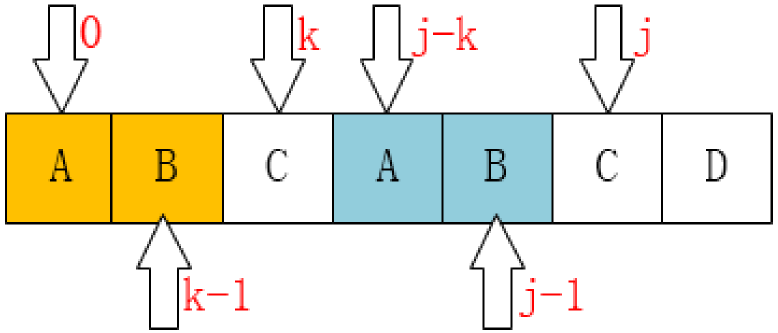
算法4-14 字符串匹配的KMP算法PatternMatchKMP(s, t)

```
输入：目标串s，模式串t
输出：返回首个有效匹配位置p，匹配失败则返回NIL
n ← s.length
m ← t.length
p ← NIL
if n ≥ m then
| GetNext (t, next)     //获得next数组   0-m-1
| i ← 0
| j ← 0
| while i<n 且 j<m do
| | if j=-1 或者 s.data[i]=t.data[j] then
| | | i ← i+1
| | | j ← j+1
| | else
| | | j ← next[j]  //i不变，j后退
| end
| if j=m then
| | p ← i-m  //匹配成功
| end
end
return p
```

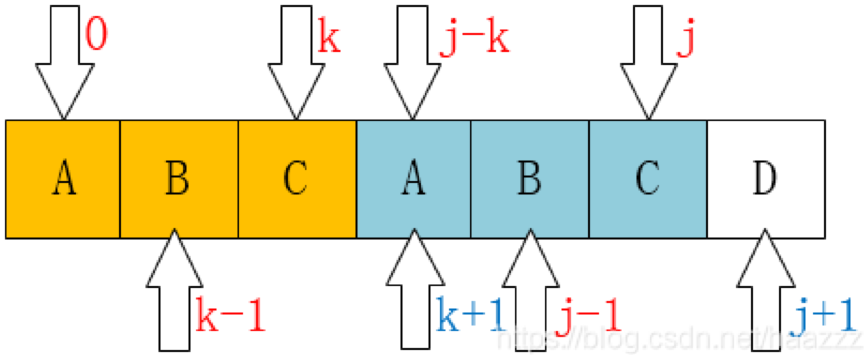
next函数值求解的算法思想

- 由定义可知next[0]=-1。利用递推法依次求j>0时的各next[j]。即已知next[0]至next[j]的值，求next[j+1]。
- 假设next[j]=k，则"$t_0 \ldots t_{k-1}$"="$t_{j-k} \ldots t_{j-1}$"。

next[j]是指**t[j]**字符前有多少个字符与**t**开头的字符相同。



- 若$t_{k}=t_j$，则"$t_0 \ldots t_{k-1}t_k$"="$t_{j-k} \ldots t_{j-1}t_j$"，根据定义，则**next[j+1]=k+1**；



next数组的求解算法：

算法4-13 next数组GetNext (t, next)

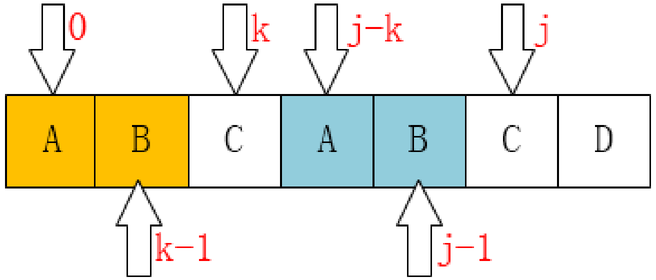
```
输入：字符串t
输出：字符串t的next数组
m ← t.length
next[0] ← -1
j ←0
k ← -1
while(j<=m-1)    //
| if k=-1 或者 t.data[j]=t.data[k] then
| |  j ← j+1;
| |  k← k+1;
| | next[j] ← k;
| else
| |  k=next[k];
| end
end
```

$T_k=T_j$

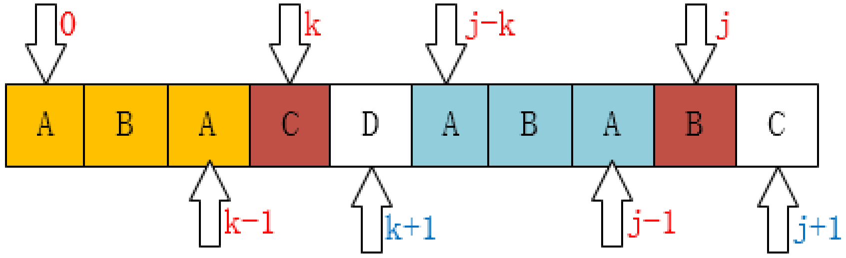
注意，此处算法**4-13**与教材有不同

next函数值求解的算法思想

- 由定义可知next[0]=-1。利用递推法依次求j>0时的各next[j]。即已知next[0]至next[j]的值，求next[j+1]。
- 假设next[j]=k，则"$t_0 \ldots t_{k-1}$"="$t_{j-k} \ldots t_{j-1}$"。



- 若$t_k=t_j$，则"$t_0 \ldots t_{k-1}t_k$"="$t_{j-k} \ldots t_{j-1}t_j$"，根据定义，则next[j+1]=k+1；
- 若$\mathbf{t_j!=t_k}$：



可将**求next函数的问题看成是一个模式匹配的**问题，整个模式串t既是主串又是子串，主串中的字符t_j和子串中的字符t_k不相等而发生了不匹配。

主串	t_0	……	t_{k-1}	……	t_{j-k}	t_{j-k+1}	……	t_{j-1}	t_j	t_{j+1}	……
子串	…	……	…	……	t_0	t_1	……	t_{k-1}	t_k	t_{k+1}	……

根据KMP算法的思想，

当模式串在t_k处发生失配，则模式串应向右滑动至**k'**=next[k]处，

若t_j与$t_{k'}$相同，则next[j+1]=**k'**+1=next[k]+1；

若不等，则模式串应向右滑动至k''=next[**k'**]处。

若t_j与$t_{k''}$相同，则next[j+1]=k''+1=next[next[k]]+1；

若不同，则模式串继续向右滑动，

……

以此类推，直至遇到t_j与某个t_k相等或k为-1为止。

next数组的求解算法：

算法4-13 求解字符串t的next数组GetNext (t, next)

```
输入：字符串t
输出：字符串t的next数组
m ← t.length
next[0] ← -1
j ←0
k ← -1
while(j<=m-1)    //
|  if k=-1 或者 t.data[j]=t.data[k] then
|  |   j ← j+1;
|  |   k← k+1;
|  |  next[j] ← k;
|  else
|  |   k=next[k];
|  end
end
```

$T_{k<>}T_j$

注意，此处算法4-13与教材有不同

next数组的求解算法：

$$
\begin{cases} next[0] = -1 \\ next[\mathrm{j}+1] = k+1 \end{cases}
$$

$k = -1$，或首次出现$\mathrm{t_j} = \mathrm{t_k}$

$k = next[\dots \mathrm{next}[\mathrm{j}]]$，$j \geq 0$

下标：	**0**	1	2	3	4
模 式 串 T：	a	a	a	b	c
k = next[j]：					

-1 0 1 2 0

算法4-13 求解字符串t的next数组GetNext (t, next)

```
输入：字符串t
输出：字符串t的next数组
m ← t.length
next[0] ← -1
j ←0
k ← -1
while(j<=m-1)
|  if k=-1 或者 t.data[j]=t.data[k] then
| |  j ← j+1;
| |  k← k+1;
| | next[j] ← k;
|  else
| |  k=next[k];
|  end
end
```

GetNext的时间复杂度：

$O(m)$

KMP算法及复杂度

- 时间复杂度：

若n为主串长度，m为子串长度。算法执行过程中，由于指针i无须回溯，主串只向右移动，比较的复杂度为O(n)。

计算next数组的复杂度为O(m)，故KMP算法的时间复杂度为**O(n+m)。**

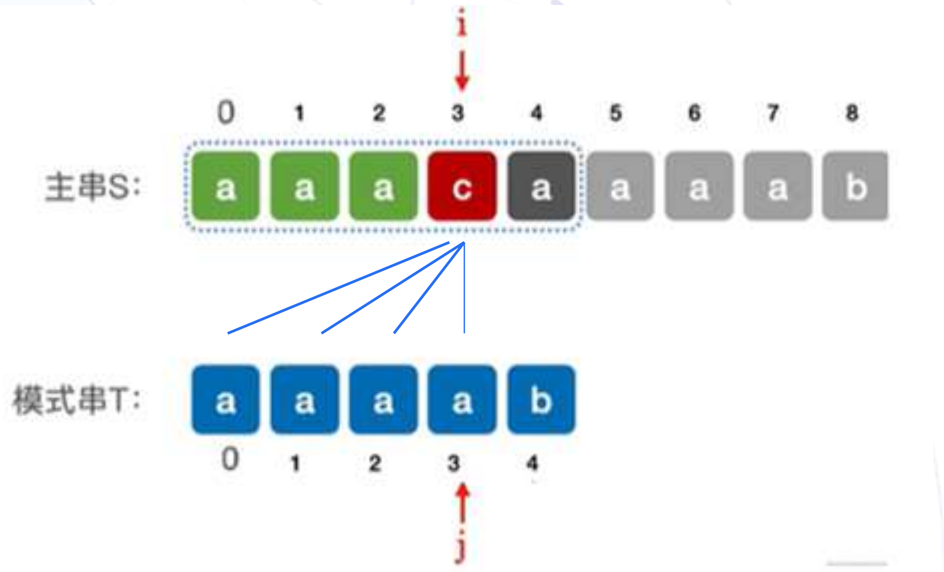
- 大多数情况下，KMP算法效率高于BF算法。
- 但当next[0]=-1，而其他next值均为0时，KMP算法退化为BF算法。

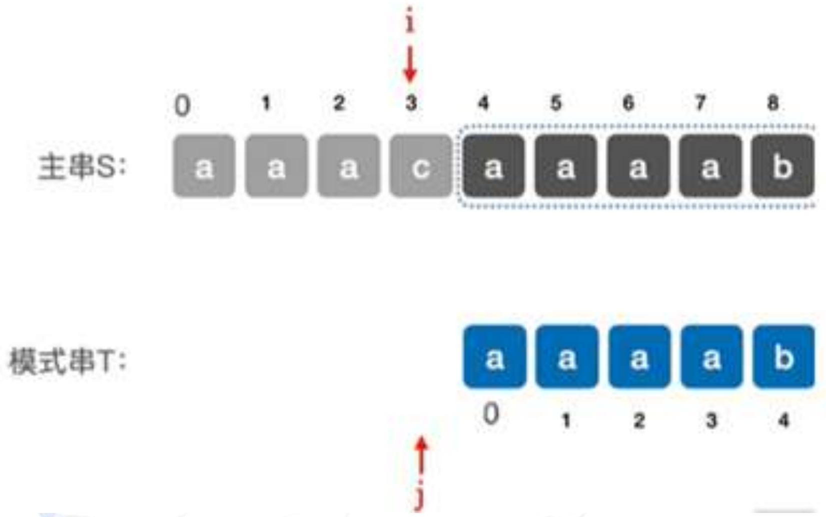
算法4-14 字符串匹配的KMP算法PatternMatchKMP(s, t)

```
输入：目标串s，模式串t
输出：返回首个有效匹配位置p，匹配失败则返回NIL
n ← s.length
m ← t.length
p ← NIL
if n ≥ m then
| GetNext (t, next)          //O(m)
| i ← 0
| j ← 0
| while i<n 且 j<m do          //O(n)
| | if j=-1 或者 s.data[i]=t.data[j] then
| | | i ← i+1
| | | j ← j+1
| | else
| | | j ← next[j]  //i不变，j后退
| end
| if j=m then
| | p ← i-m  //匹配成功
| end
end
return p
```

next函数优化

下标:	**0**	1	2	3	4
模式串 T:	a	a	a	a	b
k = next[j]:	-1	0	1	2	3
k = nextval[j]:	-1	-1	-1	-1	3





- **nextval[0]=−1**
- 当**t[*j*]=t[next[*j*]]**时：
  nextval[*j*]=nextval[next[*j*]]
- 否则： **nextval[*j*]=next[*j*]**

算法4-13 求解优化的next数组GetNext (t, next)

```
输入：字符串t
输出：字符串t的next数组
m ← t.length
next[0] ← -1
j ←0
k ← -1
while(j<=m-1)
| if k=-1 或者 t.data[j]=t.data[k] then
| |  j ← j+1;
| |  k← k+1;
| |  if t.data[j]=t.data[k] then
| ||  next[j] ← next[k]
| |  else
| ||  next[j] ← k;
| else
| |  k=next[k];
| end
end
```

next函数优化

下标：	0	1	2	3	4	5
模 式 串 T：	a	b	a	b	a	a
k = next[j]：	-1	0	0	1	2	3
k = nextval[j]:	-1	0	-1	0	-1	3

单选题 5分

设置

1. 空串与空格串是相同的。

A 正确

B 错误

提交

单选题 5分

设置

2. 字符串"abc"（ ）字符串"aabbcc"。

A 大于

B 等于

C 小于

D 无法比较

提交

单选题 5分

设置

3. 由于元素特性和操作特性，字符串通常采用顺序存储。

A 正确

B 错误

提交

单选题 5分

设置

4. 在主串"ababaababcb"中查找模式"ababc"，采用BF算法需要匹配（　）趟。

A 3

B 4

C 5

D 6

提交

单选题 5分

设置

5. 对于模式"abababab"，next[7]的值是（ ）。

A 2

B 3

C 4

D 5

提交

单选题 5分

设置

6. 在主串"ababaababcb"中查找模式"ababc"，采用KMP算法需要匹配（ ）趟。

A 3

B 4

C 5

D 6

提交

01234
ababc
-1 0 0 1 2

012345678910
ababaababcb

ababc

i=0	j=0	i=4	j=4	k=2
i=4	j=2	i=5	j=3	k=1
i=5	j=1	i=5	j=1	k=0
i=5	j=0	匹配成功		

课堂小结

字符串模式匹配的含义

BF和KMP算法